Problem 1

Let P be a set of n points in the plane, sorted on y-coordinate. Show that, because P is sorted, a priority search tree of the points in P can be constructed in $O(n)$ time.

> Since the points are already sorted, they already are in the form of a flattened binary search tree. Constructing a heap from this can be done with the standard bottom up approach and therefore will only take $O(n)$ time as normal for heaps constructed in this way.

Problem 2

Let I a set of intervals on the real line. We want to be able to count the number of intervals containing a query point in $O(\log n)$ time. Thus, the query time must be independent of the number of segments containing the query point.

Part A

Describe a data structure for this problem based on segment trees, which uses only $O(n)$ storage. Analyze the preprocessing time, and the query time of the data structure.

> The same process can be done to construct a segment tree, but with the following changes
>
> - Store a counter with every node
> - When adding an interval $[x : x']$, if $\text{Int}(v) \subseteq [x : x']$ then simply increment the counter associated to v rather than storing the interval
>
> The query then remains the same as a traversal from root to leaf, but all that needs to be done is keep a running total of the counters of all the nodes traversed. The preprocessing complexity is still $O(n \log n)$ since insertion takes $O(\log n)$ time and there are n intervals in total. The storage complexity is now $O(n)$ since there are n leaves and hence $O(n)$ of which each only need $O(1)$ space (since they no longer store intervals). The query time is also the desired $O(\log n)$ since all that needs to be done is a full root to leaf traversal with constant work at each step to keep the running total, hence $O(n \log n)$ time.

Part B

Describe a data structure for this problem based on interval trees. You should replace the lists associated with the nodes of the interval tree with other structures. Analyze the amount of storage, preprocessing time, and the query time of the data structure.

We modify the interval tree such that the nodes dont store any intervals but rather

- Store the smallest and largest endpoint of the intervals it represents
- Store how many intervals it represents

That is, instead of storing intervals at each node, we only store the range in which all of them lay in and the number of intervals in that range. The query then is to walk the tree until we reach the node that contains the query point but its children do not and report the counter stored in that node. The preprocessing time of this modified structure is still $O(n \log n)$ since $O(n)$ work still needs to be done per level of recursion to determine the lower and upper endpoints, partition based on midpoint, and count the number of intervals. Storage is clearly $O(n)$ since we are using the same number of nodes as an interval tree with $O(1)$ data per node. The query time is $O(\log n)$ because it is a simple tree traversal that may or may not be a full traversal but is still bounded by $O(\log n)$.

Problem 3

Let I be a set of intervals on the real line. We want to store these intervals such that we can efficiently determine those intervals that are completely contained in a given interval $[x : x']$. Describe a data structure that uses $O(n \log n)$ storage and solves such queries in $O(\log n + k)$ time, where k is the number of answers.

The key idea is to do a range search of the endpoints and check if an intervals endpoints were both contained in the results. To do this, construct a range tree $\mathcal{T}$ on the endpoints of the intervals in I, storing with each endpoint both the interval and the interval's index from within I. The query then is

Algorithm 1 QuerySuperset($\mathcal{T}$: Range Tree, $[x : x']$: Interval)

1. Do a range query on $\mathcal{T}$ to get all the endpoints of the intervals contained in $[x : x']$
2. Create a boolean array with n slots where n is the number of intervals as well as an empty dynamic list of intervals
3. For each endpoint from the range query
    - Mark the slot at the endpoints associated index as true/present
    - If the slot already was marked, add the interval to the dynamic list
4. Return the list of intervals

Problem 4

Again we have a collection I of intervals on the real line, but this time we want to efficiently determine those intervals that contain a given interval $[x : x']$. Describe a data structure that uses $O(n)$ storage and solves such queries in $O(\log n + k)$ time, where k is the number of answers.

We solve this by constructing a priority tree on the endpoints of the intervals, but we make sure to associate with each endpoint the interval it comes from and the index of that interval within the list I as well as if its the right or left endpoint. The query is then

Algorithm 2 QuerySubset($\mathcal{T}$: P.S.T, $[x : x']$: Interval)

1. Create a tracker array A of n booleans
2. Create an empty dynamic result list L of intervals
3. Do the PST query $(-\infty, x]$. For only the endpoints that are marked as left endpoints, mark the entry in A at the associated index as true
4. Do the PST query $[x', \infty)$. For only the endpoints that are as right endpoints, check if the entry in A at the associated index has been marked as true. If so, add the associated interval to L
5. Return L

Problem 5

Part A

Prove that the proposed data structure correctly solves range queries.

Proof. Let $p, q \in P(v)$ for some node v. Assume that p is in the query $[x : x'] \times [y : y']$ but q is not. Since every associated tree will have the bounds $[y : y']$, if q_y is not in this interval it will be rejected. Suppose then WLOG that q_y is in this interval and that $q_x < x$. Then it is less than the x-coordinate of the split node. Therefore $q \in \mathcal{T}_{\text{left}}$ of $lc(v)$ which will be rejected by the query $[x : \infty)$. It is clear that p will be detected by the query since the main query is the same as the union of the two unbounded queries of the children nodes. Therefore p will be detected in the query and q will not. ■

Part B

What are the bounds for preprocessing time, storage, and query time of this structure? Prove your answers.

Proof.

Preprocessing

Constructing the initial search tree on the x-coordinates takes $O(n \log n)$ time. If h is the depth (starting from 0) of some node v in this tree, then $P(v)$ will have $O\left(\frac{n}{2^h}\right)$ nodes. Constructing both of the priority search trees will then take

$$O\left(\frac{n}{2^{h+1}} \log\left(\frac{n}{2^{h+1}}\right)\right)$$

for a node at depth h. Since there are 2^h nodes at depth h, we have a total running time of

$$\begin{aligned}
\sum_{h=0}^{\lceil \log n \rceil} 2^h O\left(\frac{n}{2^{h+1}} \log\left(\frac{n}{2^{h+1}}\right)\right) &= \sum_{h=0}^{\lceil \log n \rceil} O\left(n \log\left(\frac{n}{2^{h+1}}\right)\right) \\
&= \sum_{h=0}^{\lceil \log n \rceil} O\left(n\left(\log(n) - \log\left(2^{h+1}\right)\right)\right) \\
&= \sum_{h=0}^{\lceil \log n \rceil} O(n(\log(n) - (h+1)\log(2))) \\
&= n \sum_{h=0}^{\lceil \log n \rceil} O(\log(n) - (h+1)\log(2))) \\
&= n\left((\log n) O(\log n) - O(\log^2 n)\right) \\
&= nO(\log^2 n) = O(n \log^2 n)
\end{aligned}$$

Hence the preprocessing time is $O(n \log^2 n)$.

Query

Finding v_{split} only takes $O(\log n)$ time like other similar range structures. The PST queries in each subtree take then $O(\log n + k_1)$ and $O(\log n + k_2)$ time where $k_1 + k_2 = k$, hence all together the query takes only $O(\log n + k)$ time.

Storage

A node at depth h stores $2 \cdot \frac{n}{2^{h+1}} = \frac{n}{2^h}$ points. Since there are 2^h nodes at each depth, there are $2^h \cdot \frac{n}{2^h} = n$ points stored. Since the height of the tree is $\log n$ we have a storage complexity of $O(n \log n)$. ■